Chapter 1

Modularity and Neural Networks

Contributed by Mark Horeni

1.1 Introduction

Community detection is useful unsupervised way to understand more information about a graph. One way to do community detection is by maximizing a global property of the graph known as modularity. Maximizing modularity has been used in a wide variety of applications with some success in not only biological networks, but other social networks and beyond for community detection [11] [14]. Specifically, modulairty maximization techniques have been shown to out perform other community detection algorithms [14].

A lot is known about the human brain, but seemingly nothing is known about it. To study the human brain scientists typically look at different, more simple examples of connections between neurons, also known as *connectomes*. Mapping and knowing the functionality of connectomes is a hard problem because someone has to look at when a stimulus is received, what neurons fire when and where. Since these are structures of neurons, it seems like a good assumption that these neurons would form topologically dense communities in order to send information where it needs to go.

Finding communities of neurons would aid to our knowledge about the brain. Knowing that certain neurons work together to perform a function would help us understand interactions of drugs better, along with insights into the philosophy of knowing ourselves better.

1.2 The Problem as a Graph

Individual neurons can be thought of as nodes, and each neuron has two types of connectors, either gap junctions or chemical synapses[15]. In the *C. elegans* round worm, chemical synapses as seen in Figure 1.1, can have 1, 2, or 3 directed outputs to another neuron, while similarly, gap junctions can have multiple outputs, but these outputs are undirected as the electrical flow can flow either way theoretically, even though in a given function this does not end up happening [15].

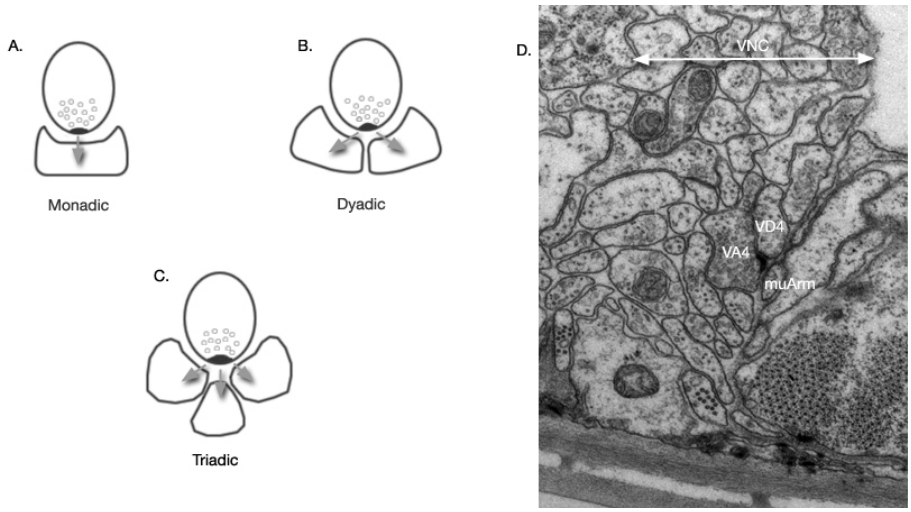

Figure 1.1: A view of the neurons and their chemical synapses [4]

1.3 Some Realistic Data Sets

The *c. elegans* is a transparent roundworm that has had all of its neurons mapped along with all of the connections. There are a total of 279 neurons, and between them there are around 6393 chemical synapses and 890 electrical gap junctions[4]. Each neuron has an attribute of whether the neuron itself is either a motor, sensory, or inter neuron (or a combination of), and the distribution of those are roughly equal across neurons [4].

The worm data is the only complete data, but there does exist partial data for other animals including partial data from flies, cats, macaques, mice, rats, and humans.For this project the connectome of a mouse retina, which is 1123 neurons and 577350 connections between neurons [9] will be used as a benchmark for the middle. Also, MRI data gathered from patients and then transformed into a graph will also be used as a benchmark for a large graph. This human data consists of 277,345 neurons and around 64.4 million connections between each of these neurons [3].

1.4 Louvain-A Key Graph Kernel

Modularity, Q, is a measure of how dense a community is compared to how dense a community is expected to be. The assumption in using modulairty maximization is that nodes inside communities are more densely connected to each other than to any other node in the network. This is defined as the following [5]

$$Q = \frac{1}{2m}\sum_{i,j}[A_{ij} - \frac{k_i k_j}{2m}]\delta(c_i, c_j)$$

where 2m is the weight of all edges, Aij is the weight between i and j, k_i and k_j are the total weights attached to each i and j, and c_i and c_j are the communities. The goal is to find which combinations of nodes when grouped into certain communities, which combination maximizes modulairty.

1.4.1 Undirected Louvain

Since the goal is to maximize modulairty, the approach of the Louvain algorithm is greedy optimization. To do this, the algorithm first starts with every node in its own community [5]. Next,

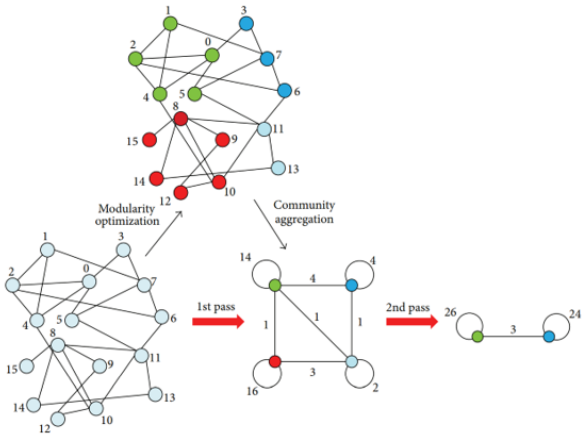

Figure 1.2: Example of the Louvain Algorithm [10]

each node is put into a neighboring community and the change in modularity is calculated by

$$\Delta Q = [\frac{\sum_{in} + k_{i,in}}{2m} - (\frac{\sum_{tot} + k_i}{2m})^2] - [\frac{\sum_{in}}{2m} - (\frac{\sum_{tot}}{2m})^2 - (\frac{k+i}{2m})^2]$$

where $\sum_{in}$ are the total weights inside community C, $\sum_{tot}$ is the sum of edges of the links incident to nodes in C, k_i are the sum of incident links of node i, and $k_{i,in}$ is the sum of weights from i in C with m being the total sum of weights in the network [5].
The other half of the algorithm takes the previous phase, and turns each community into its own node with a self loop with the weight of all the edges of all the nodes inside the community. When this finishes, the process goes back to the first half, and the process is repeated until modularity no longer increases between iterations. This proccess is shown visually in figure 1.2.

1.4.2 Resolution Limit

One of the problems with maximizing modulairty is that there is a problem known as the "resolution limit". Since the formula for modularity is a global property and uses the strength within a community compared to the strength between communities, in some networks the strength between real communities may be so close to the strength between communities that modularity may be optimized if a strong link between two communities is joined into one community [7].

A solution proposed to solve this problem is to introduce a time-scale parameter t to help stabilize modularity optimizing the quality function [12]

$$Q_{NL}(t) = (1-t) + \frac{1}{2m}\sum_{i,j}[A_{ij}t - \frac{k_i k_j}{2m}]\delta(c_i, c_j)$$

.

1.4.3 Directed Louvain

Although modularity is usually defined for unweighted graphs, in directed graphs it can be defined as

$$Q_d = \frac{1}{m}\sum_{i,j}[A_{ij} - \frac{d_i^{in} d_j^{out}}{m}]\delta(c_i, c_j)$$

where the only difference is that m is now the weight of all the arcs (directed edges), and d^{in} stands for the in degree of i while d_j^{out} stands for the out degree of j [6].
Similarly, change in modularity can be defined as

$$\Delta_{Q_d} = \frac{d_i^C}{m} - [\frac{d_i^{out}\sum_{tot}^{in} + d_i^{in}\sum_{in}^{out}}{m^2}]$$

where $\sum_{tot}^{in}$ is the sum of all in-going arcs into community C, and $\sum_{tot}^{out}$ are all the out-going arcs out of community C [6].

1.4.4 Psuodcode

The pusdocode of the algorithm appears to run with time complexity $O(n \log n)$, because at every step nodes are guaranteed to join a community, meaning the number of communities will decrease every time giving it a $\log n$ appearance. Though it can be argued that the time complexity is actually closer $O(n^2)$ because if only 1 node joins a community at a time, then the algorithm has to run n times for n number of nodes. The psudocode is as follows [11]

Algorithm 1 Louvain

V: a set of vertices
E: a set of edges
W: a set of weights of edges, initialized to 1
$G \leftarrow (V, E, W)$
repeat
  $C \leftarrow \{\{v_i\}\}|v_i \in G(V))\}$
  Calculate current modularity Q_{cur}
  $Q_{new} \leftarrow Q_{cur}$
  $Q_{old} \leftarrow Q_{new}$
  repeat
    for $v_i \in V$ **do**
      $Q_{new} \leftarrow Q_{cur}$
      remove v_i from its current community
      $N_{v_i} \leftarrow \{c_k|v_i \in G(V), v_j \in c_k, e_{ij} \in G(E)\}$
      find $c_x \in N_{v_i}$ that has $max\Delta Q_{\{v_i\},c_x} > 0$
    end for
    Calculate new modularity Q_{new}
  until no membership change or $Q_{new} = Q_{cur}$
  $V' \leftarrow \{c_i|c_{\in}C\}$
  $E' \leftarrow \{e_{ij}|\forall e_{ij}$ if $v_i \in C_i, v_j \in C_j$, and $C_{\not=} C_j$
  $W' \leftarrow \{w_{ij}|\sum w_{ij}, \forall e_{ij}$ if $v_i \in C_i$ and $v_j \in C_j\}$
until $Q_{new} = Q_{old}$

This psudocode starts out by accepting $\{V, E, W\}$ which are the sets of vertices edges and weights, and then loops calculating current modularity by moving node v_i into its neighboring communities. This loop will stop when all the nodes stop changing communities or if modularity stops increasing. Finally, a new graph is created. The new set of vertices V' is now the communities found. The new set of edges E' are the edges that existed between nodes before they were put into a community, including self loops for intra-community connections. The new set of weights W' are the summation of the weights transformed from changing the edges from nodes to communities.

1.5 Prior and Related Work

As mentioned previously in this paper, community detection has been applied to connectome data before. The focus on most of this research is on the *C. elegans* database because there exists a lot of metadata for this data set, therefore ground truth on most questions can be known. This hasn't prevented from experimenting on other larger connectome graph though, just to a lesser degree of clarity [10][11][14].

A lot of work has also been done on community detection as a whole, and much on specifically Louvain. The whole idea of the Louvain kernel came out the need for wanting faster community detection [5]. Exploration of different attributes of a graph were explored, such as directedness [6]. This idea was then taken even further by parallelizing it, and using different heuristics, made the algorithm even faster [13].

1.6 A Sequential Algorithm

The initial sequential algorithm can be simply generated by following the psudocode in section 1.4. The implementation in the next section uses dendograms [2] for processing and NetworkX uses hash tables [1] for the storage of graphs, although this isn't the most efficient, it is simple and easy to implement.

1.7 A Reference Sequential Implementation

For my implementation, I used NetworkX (python), as there already existed a library that implemented Louvain [2], but this library did not support directed graphs, so I had to change the definition of modularity from the original. The main thing that was changed that it now uses both in and out degrees, instead of just total degree and dividing by 2. The original implementation stored the vertex degree pairs as a dictionary and would square the degree. I rewrote these few lines of code so instead of just using the total degree and dividing by 2, it instead uses the in degree and the out degree.

1.8 Sequential Scaling Results

Since the worm database was small enough, this was able to run on my computer in under a second, though measurements in this paper are based on data collected from running the implementation on Notre Dame's Center for Research Computing cluster. The parameter that was varied was t, the resolution value, explained in 1.4.2, along with the three databases mentioned earlier in the paper. The results are displayed in figure 1.3. As can be seen, as the resolution value goes down, the time also goes down, and in the Mouse Retina database, quite significantly. The final database of a subset of human data did not finish running, as I killed it after 12 hours and decided to save it purely for the enhanced implementation.

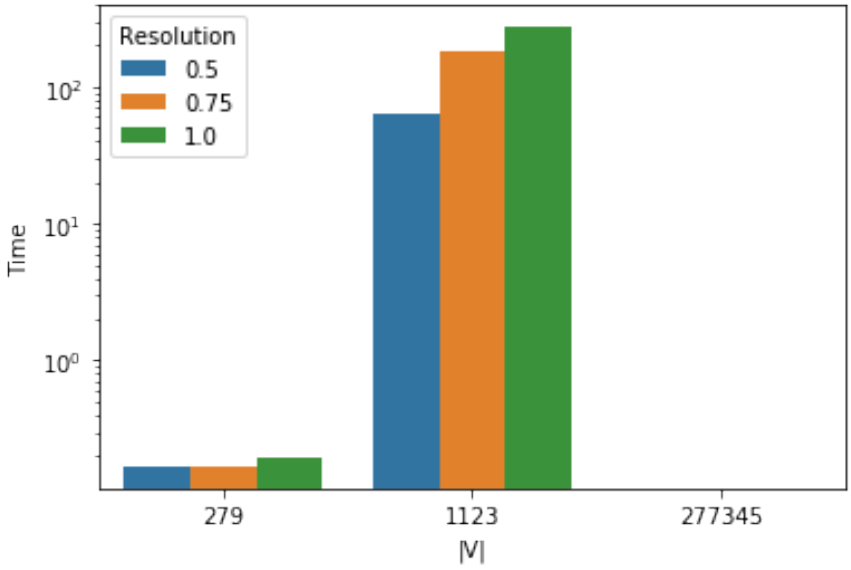

Figure 1.3: Runtime as resolution value decreases

1.9 An Enhanced Algorithm

A algorithm for a parallelized version Louvain was proposed back in 2015 [13]. The algorithm is very basic in theory, as shown by the psudocode in Algorithm 2 [8]. The algorithm consists of two main parts, Louvain Iterations and building the graph for the next step.

Algorithm 2 Louvain Parallel

$C_{curr} \leftarrow \{\{u\} | \forall u \in V\}$
$\{currMod, prevMod\} \leftarrow 0$
while true **do**
  $currMod \leftarrow LouvainIteration(G_i, C_{curr})$
  if $currMod - prevMod \leq \tau$ **then**
    break and output the final set of communities
  end if
  $BuildNextPhaseGraph(G_i, C_{curr})$
  $prevMod \leftarrow currMod$
end while

1.9.1 Louvain Iterations

The first part of the proposed parallel algorithm is as described in Algorithm 3 [8]. This algorithm takes in a subgraph G_i, and performs the same thing as the sequential Louvain would do, except on the subgraph. After this is done, the nodes calculate the modularity for their subgraph and reduce all of the modualrities from all the subgraphs to get the current modularity of the graph.

Algorithm 3 Louvain Iteration

$V_g \leftarrow ExchangeGhostVerticies(G_i)$
while true **do**
  send the latest information on those local vertices that are stored as ghost vertices on remote processes
  recieve latest information on all ghost verticies
  for $v \in V_i$ **do**
    Compute ΔQ by moving v to neighboring communities
    Determine which community v belongs to by maximizing ΔQ, and update community information
    Send updated information about ghost vertices to owner processes
    $C_{info} \leftarrow$ receive and update information on local communities
  end for
  $cuurMod_i \leftarrow$ Compute modularity based on the current subgraph and the community information C_{info}
  $currMod \leftarrow$ all-reduce $\sum_{\forall_i} currMod_i$
  if $currMod - prevMod \leq \tau$ **then**
    break
  end if
end while
return $prevMod \leftarrow currMod$

1.9.2 Building the Next Graph

This step in the algorithm is very similar to the step in the sequential algorithm. In the sequential algorithm the new graph is generated by taking the communities, making them into vertices, and making replacing the edges with their weights as the summation of their previous connections.

In this algorithm, the same is done on a local scale, where nodes are remapped from their old communities (or the starting vertex) to their new ones. The next part of the algorithm is to look at all these local processes, and globally number all the *unique* communities. The edge lists of the communities are then combined, where if an edge already existed between processes the weights would be added, or if an edge didn't exist then that edge would be appended to the edge list.

1.10 A Reference Enhanced Implementation

The makers of the original algorithm created a software package called *Grappolo*. This is a library written in C++ that uses OpenMP. The there are about 500 lines of code that actually have to do with the algorithm, and a bunch for for stuff like I/O. The library uses CSR's to store the graph instead of Python's dictionaries, so the amount of memory used should be a lot less.

A modification was made so that testing of different resolution values similar to the sequential implementation could be done. This ended up being editing of 1 line of code, making sure that the modularity function optimized the new quality function mentioned in 1.4.2.

1.11 Enhanced Scaling Results

The measurements are based off of using 24 cores running on the Notre Dame's Center for Research Computing cluster. Included in this section are both the wall-clock run time of each dataset given

the resolution value in Figure 1.5 along with how many times the function "Louvain Iteration" was called in Figure 1.4, to see how many iterations there actually were.

The total time the longest graph took was around 30 seconds, which is significantly faster than the more than 12 hours it took in the sequential results. As the resolution limit decreased, there were less iterations, the algorithm ran faster, which would make sense and shows the same trend as in the sequential implementation.

As the amount of nodes increased, the time also went up. This seemed to increase $O(n \log(n)$ fashion, but as mentioned before it seems safer to say $O(n^2)$ as these graphs weren't designed to test worst case, only real world practicality.

The enhanced algorithm ran more than 24 times faster, even though only 24 cores were used. This seems to be mainly because since louvain is a greedy algorithm, and since every process has its own subgraph in charge of nodes optimizing a local modularity, a dense subgraph is more likely to be found locally than globally.

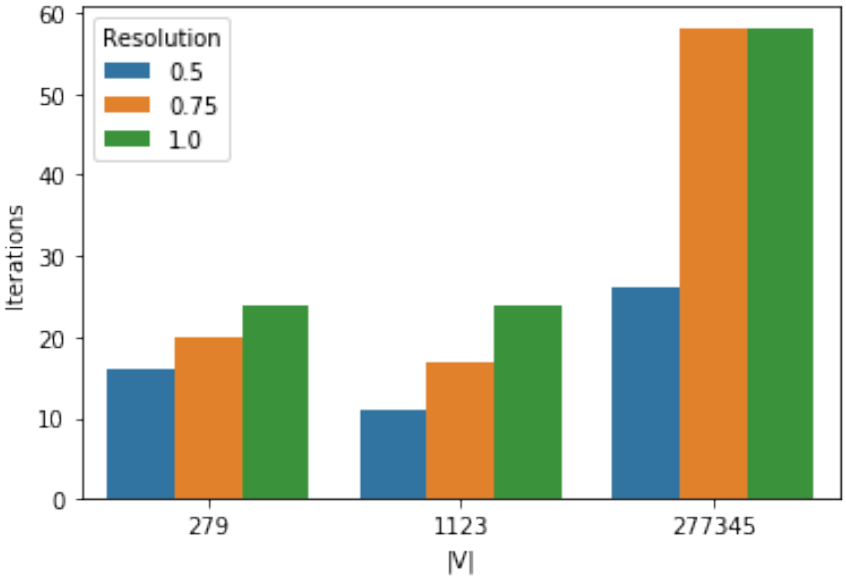

Figure 1.4: Number of iterations taken to converge to maximum Modularity

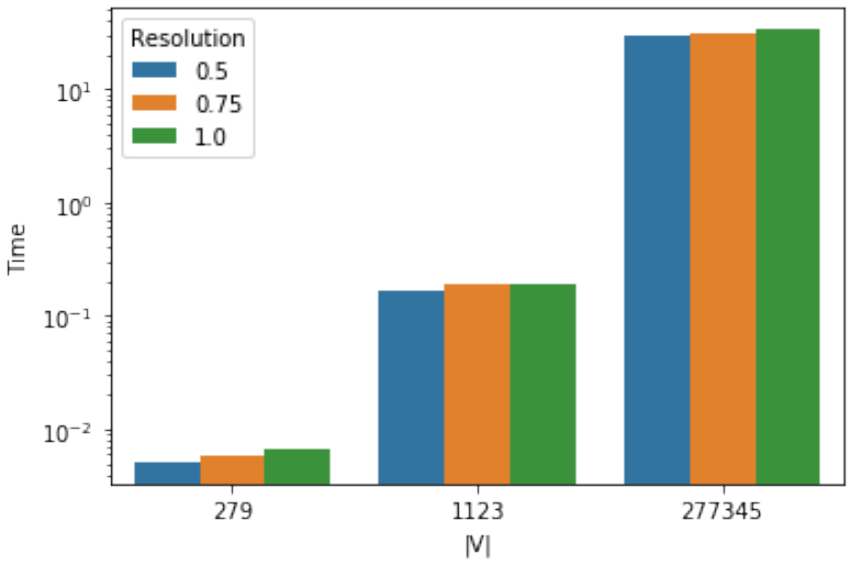

Figure 1.5: Number of seconds taken to converge to maximum Modularity

1.12 Conclusion

Although in theory communities may be topologically dense, this does not to seem much in real life. With some metadata analysis of the worm data set with the communities given, there didn't seem to be a strong correlation between anything. This should be researched further to see if there's some combination of meta data that can give positive results, or maybe the topology of the graph means nothing at all.

Louvain as an algorithm is a step in the right direction, as it is faster and easily parallelized to give an even bigger speedup. A problem though is that this speedup comes at the cost of stability, the nodes in each community seemed to jump almost randomly from community to community because of the slightest increase in modularity, even if it didn't seem to be the best fit. Future work should look at the stability of Louvain to keep the speedup while maintaining quality.

1.13 Response to Reviews

Once again more detail was added into previous sections, along with some clarification. Actual code for sequential implementation was taken out because it was just taking up space and didn't really add anything I couldn't say in a few sentences. Some grammar and spelling mistakes throughout the paper were also fixed.

Bibliography

[1] https://networkx.github.io/documentation/stable/.

[2] https://python-louvain.readthedocs.io/en/latest/.

[3] Katrin Amunts, Claude Lepage, Louis Borgeat, Hartmut Mohlberg, Timo Dickscheid, Marc-Étienne Rousseau, Sebastian Bludau, Pierre-Louis Bazin, Lindsay B. Lewis, Ana-Maria Oros-Peusquens, Nadim J. Shah, Thomas Lippert, Karl Zilles, and Alan C. Evans. Bigbrain: An ultrahigh-resolution 3d human brain model. *Science*, 340(6139):1472–1475, 2013.

[4] Beth Li Ju Chen Beckman. Neuronal network of c . elegans : from anatomy to behavior. 2007.

[5] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, oct 2008.

[6] Nicolas Dugu and Anthony Perez. Directed louvain : maximizing modularity in directed networks, 2015.

[7] Santo Fortunato and Marc Barthélemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1):36–41, 2007.

[8] S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, H. Lu, D. Chavarri-Miranda, A. Khan, and A. Gebremedhin. Distributed louvain algorithm for graph community detection. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 885–895, May 2018.

[9] Moritz Helmstaedter, Kevin L. Briggman, Srinivas C. Turaga, Viren Jain, H. Sebastian Seung, and Winfried Denk. Connectomic reconstruction of the inner plexiform layer in the mouse retina. *Nature*, 500(7461):168–174, aug 2013.

[10] Yong-Hyuk Kim, Sehoon Seo, Yong-Ho Ha, Seongwon Lim, and Yourim Yoon. Two applications of clustering techniques to twitter: Community detection and issue extraction. *Discrete Dynamics in Nature and Society*, 2013:1–8, 2013.

[11] Haewoon Kwak, Yoonchan Choi, Young-Ho Eom, Hawoong Jeong, and Sue Moon. Mining communities in networks: A solution for consistency and its evaluation. pages 301–314, 01 2009.

[12] Renaud Lambiotte, Jean-Charles Delvenne, and Mauricio Barahona. Random walks, markov processes and the multiscale modular organization of complex networks. *IEEE Transactions on Network Science and Engineering*, 1(2):76–90, jul 2014.

[13] Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. Parallel heuristics for scalable community detection. *Parallel Computing*, 47:19–37, aug 2015.

[14] M. E. J. Newman. Modularity and community structure in networks. *Proc Natl Acad Sci U S A*, 103(23):8577–8582, Jun 2006. 2388[PII].

[15] J. G. White, E. Southgate, J. N. Thomson, and S. Brenner. The structure of the nervous system of the nematode caenorhabditis elegans. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 314(1165):1–340, nov 1986.